Module 4: Trusted Execution Environments

Task 1: Isolated Execution [25 Points]

Prof. Shweta Shinde *shweta.shivajishinde@inf.ethz.ch*
Supraja Sridhara *supraja.sridhara@inf.ethz.ch*
Mark Kuhne *mark.kuhne@inf.ethz.ch*

Welcome to the fourth module of the Information Security Lab. This task is part of the Module-04 on *Trusted Execution Environments*, or TEEs in short.

1 Overview

TEEs, for example Intel SGX, isolate user-space programs in secure *enclaves* that do not trust the operating system. TEEs protect program execution inside an enclave by assuming hardware support for several operations, including memory isolation. The hardware-based attestation can be used to setup shared secrets between entities to build secure channels.

1.1 Task Summary

This task explores the importance of enclave-based isolated execution of security-sensitive logic. For this task, we will consider a secure password storage service as our example. In our setup, a Remote Party RP uses a secure password storage service to store and update passwords. Figure 1 shows this setup. The password storage service is made up of several secure enclaves: a Manager M, a String Parser SP, and a secure storage Peripheral P. M receives encrypted messages from RP, parses them, and stores them in the secure storage peripheral P. Since the task of parsing the packet is non-sensitive, M delegates this task to a separate String Parser enclave SP.

Enforcing Isolation. For this task, we don't have access to a hardware based TEE. Instead, we have emulated the isolation by making the components (RP, M, SP, P) black-box and non-debuggable. In other words, you do not have `sudo` access and thus cannot perform any operations that a compromised OS could. So, for all practical purposes for this task, these user processes are considered to be secure enclaves. Note that, for simplicity, we run all the components on the same machine.

Attacker Capabilities. Under the above setting, you (the attacker) cannot do much. However, we have given you one ability—you have `r-w-x` permissions for the SP process.

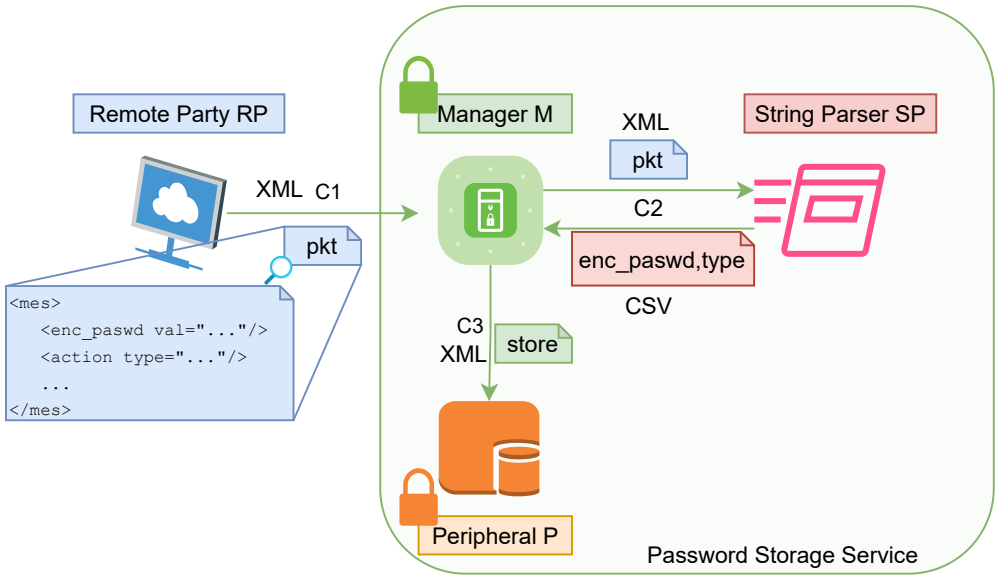

Figure 1: Overview of the password storage service. The password Manager M receives the encrypted password enc_paswd and action $action$ from a remote party RP over a secure channel C1, as an XML message. M sends the message to a string parser SP over another secure channel C2. The string parser SP parses the message from M and returns back a comma-separated string $\{\langle enc\_paswd \rangle, \langle type \rangle\}$ over channel C2. M then performs computation based on $type$ and stores the result in a secure storage peripheral P. For this, it sends a $store$ encrypted message over a secure channel C3 to P. We assume that the secure channels C1, C2, C3 are pre-established and are not vulnerable to attacks over the network.

Goals. The goal of this task is to understand the power of an attacker who has the above capabilities. Specifically, you will be performing the following sub-tasks:

Task 1.1 Implement an exploit that breaks the secure channel C2 between M and SP.

Task 1.2 Implement an exploit that changes the runtime execution of SP.

2 Getting Started

For this task you should download VM_Task1.ova from Moodle and create a new virtual machine (VM) in VirtualBox. For more details on how to download and install VirtualBox and create a new VM from the .ova refer to the exercise handout on Moodle. We strongly recommend that you use the default configuration (RAM, CPU, and network) in the .ova file. This is important so as to ensure that the timing measurements you do locally (e.g., total execution time) will be the same on our grading VM. We will use the same VM configurations to grade your solutions.

2.1 Testing your setup

Once you have created the VM from VM_Task1.ova and started it in VirtualBox, you can boot the VM. To test your setup follow these steps:

Listing 1: Directory structure of VM_Task1.ova

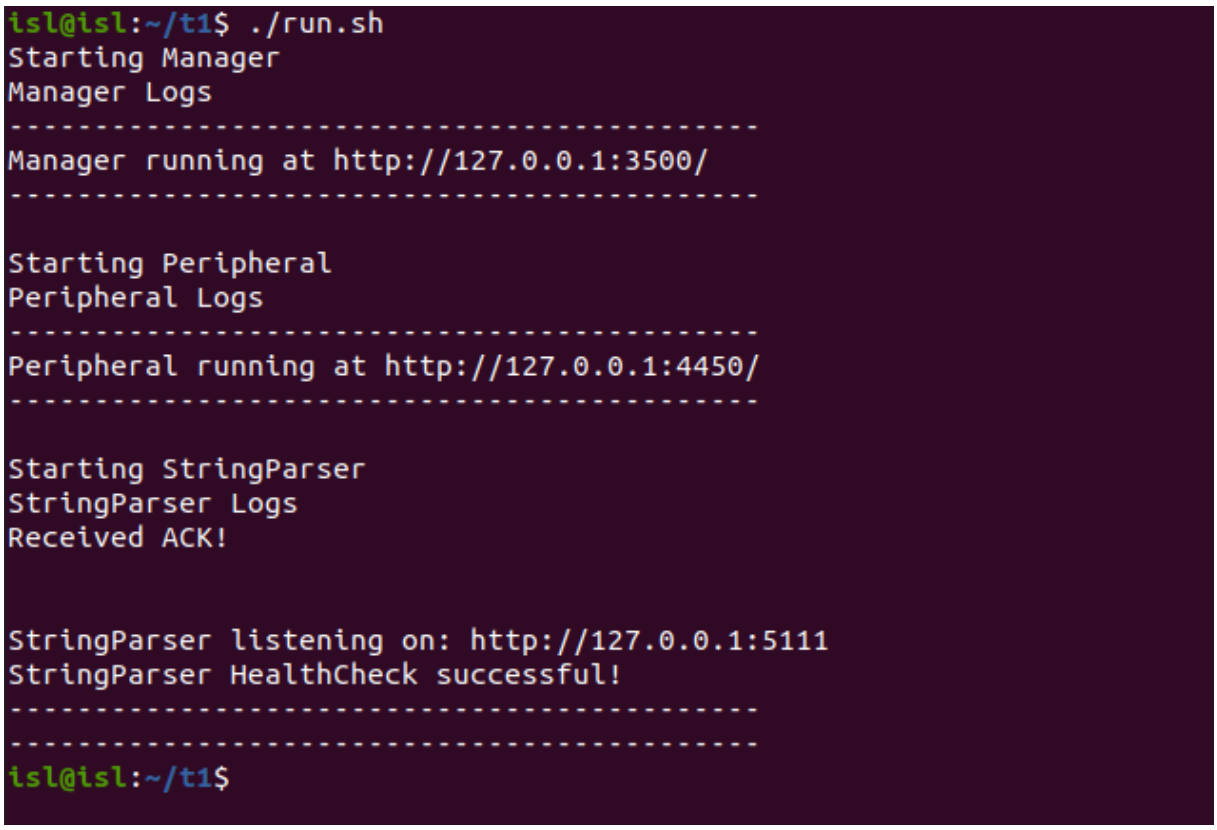

Figure 2: Example execution of the `run.sh` script

```
isl/
├── user
├── scripts/
└── t1/
    ├── manager
    ├── peripheral
    ├── remote_party
    ├── run_manager.sh
    ├── run_peripheral.sh
    ├── run_string_parser.sh
    ├── string_parser
    ├── test_setup.py
    ├── test_manager.py
    ├── test_peripheral.py
    ├── test_string_parser.py
    ├── start.sh
    └── run.sh
```

1. Execute `cd /home/isl && tree -L 2`. You should see the folder structure shown in Listing 1. Each directory and file in this tree has a specific purpose, as explained below.

   (a) `user`: this file contains your ETH id e.g., `11-111-111`. The value from this file is read by different binaries to generate user-specific flags.

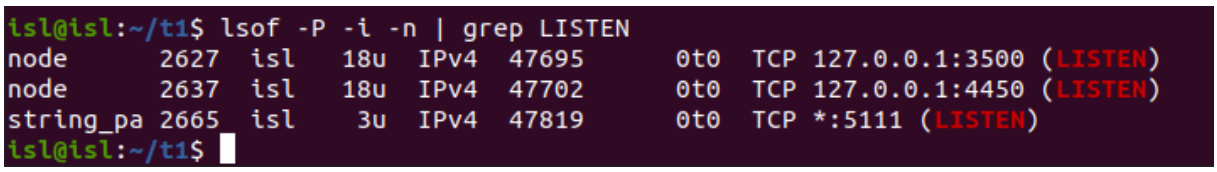

Figure 3: M and P are node processes listening on ports 3500 and 4450 respectively. SP listens on port 5111

(b) `scripts`: this folder is empty. The grading script runs your solution script from this folder. See Section 4.4 for more details on the grading scripts.

(c) `t1`: this folder contains all binaries and scripts required for this task.

- `manager`, `peripheral`, `remote_party`, `string_parser`: Executable binaries for M, P, RP, SP respectively. These are binaries i.e., you do not have access to the source code. Your solution script might not pass in our grader if you replace/modify these binaries, see Section 4.3 for details.
- `test_setup.py`, `test_manager.py`, `test_peripheral.py`, `test_string_parser.py`: these python scripts are used to test your setup and debug it in case of issues. Read Section 2.2 for more details on how to use these scripts.
- `start.sh`: starts RP and triggers communication between the various entities shown in Figure 1. For more details, see Section 3.2. Not that, this script does not kill an already running RP process.
- `run.sh`: kills any running M, P, RP, and SP processes and starts M, P, and SP processes.
- `run_manager.sh`, `run_peripheral.sh`, `run_string_parser.sh`: can be used by your solution script to individually start the components; Note that, these scripts do NOT automatically kill any running M, P, RP and SP before executing.

2. Execute `./t1/run.sh`. This script starts M, P, and SP. If the processes are started correctly, the script displays a success message and lists the open ports for all the components as shown in Figure 2. If the script fails, please read Section 2.2 to debug your setup.

3. Execute `lsof -P -i -n| grep LISTEN`. You should see M and P as node processes and SP as a `string_parser` process listening on the ports, as shown in Figure 3.

2.2 Debugging your setup

1. If executing `/home/isl/t1/run.sh` fails, ensure that the ports needed by the application are not held by any other processes. To check processes listening on a port, execute `lsof -i:<port_number>`. We have ensured that M, P, SP, and RP use distinct port numbers. If you have installed or written programs that use these ports, please modify them to use other ports. To kill processes that are using the port, execute `kill -9`

`$(lsof -t -i:<port_number>)`. Replace `<port_number>` with the port number you are interested in.

2. If you do not see any messages in the terminal, despite triggering the communication by running the `/home/isl/t1/start.sh`, check that the components are running correctly. For this, you can run the `test_setup.py` script which tests all the components and reports their status. To investigate the components individually, run the corresponding test script in `/home/isl/t1` folder.

   Individual test scripts `test_*.py` send a healthcheck message to the port that the component listens on. The component responds with a success or failure message. The response messages are printed to the console and should help you identify the problem with the component.

3. If any of the components hang or crash because of your experimental executions, run the `/home/isl/t1/run.sh` script and follow the steps in Section 2.1 to test your setup.

4. Please note that the programs do not have any persistent state. So, if you restart them, your execution state will not be stored. You will have to redo any actions you had performed before the restart.

3 Task Description

3.1 Setup

As shown in Figure 1, for this task we consider 4 components—a Manager M, a secure storage peripheral P, a remote party RP, and a string parser SP. These entities communicate using HTTP packets. However, note that it is over secure channels C1, C2, and C3. In this task, we assume that these secure channels cannot be compromised. So, even though it is not over HTTPS, a network adversary cannot compromise the channels. M, P, and RP run in secure enclaves and their execution cannot be compromised by an adversary defined in Section 1.1. SP runs in a secure enclave too, but is potentially vulnerable to the adversary from Section 1.1.

3.2 Communication

The communication between the entities proceeds as depicted in Figure 4. The HTTP POST requests are sent to http://127.0.0.1:$\langle port \rangle$. All messages are encrypted and integrity protected using AES-GCM. Each of the messages has a specific format which is explained in Section 3.3. The messages are encapsulated in a packet as explained in Section 3.4.

3.3 Message format

Listings 2–6 show the formats for the messages in Figure 4. All the listings are in XML format, except for Listing 3 which is a comma separated string. When an attribute has value `"*"`, it can be any string and is not constrained. However, the message length cannot be more than

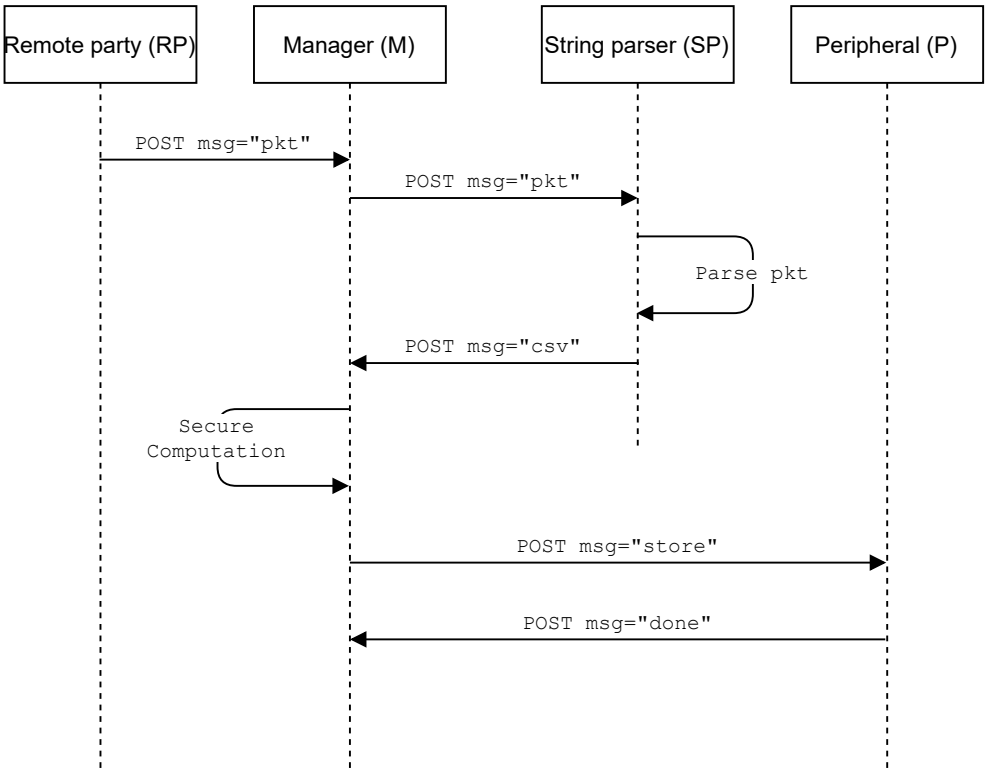

Figure 4: Communication between components. For simplicity we only show the message type and not the body of the message.

570 bytes. If the attribute values are listed in curly braces, for example `{'str1', 'str2', 'str3'}`, then the value can be only one of the strings—`'str1'` or `'str2'` or `'str3'`. Note that the white-spaces in the Listings below are to improve readability. The components send the XML's without white-spaces.

Listing 2: Message type : pkt

```
1 <mes>
2    <enc_paswd val="*"/>
3    <action type="{store, update, delete}"/>
4    <token val="*"/>
5    <timestamp val="*"/>
6    <id val="*"/>
7 </mes>
```

Listing 3: Message type : csv

```
1 enc_paswd,type
```

In Listing 3, the `enc_paswd` is the value of the encrypted password (Line 2 of Listing 2) from `pkt` and the `type` is the value of the `action` element (Line 3 of Listing 2) in `pkt`.

Listing 4: Message type : store

```
<mes>
   <enc_paswd val="*"/>
   <action type="store"/>
   <token val="*"/>
   <timestamp val="*"/>
   <id val="*"/>
</mes>
```

Listing 5: Message type : done

```
<mes>
   <action type="done"/>
   <token val="*"/>
</mes>
```

In addition to the above message types, the components can send `key-update` messages (see Listing 6). This message triggers a key update for the secure channel between the sender and the receiver. For example, if M sends the `key-update` message to P the key used for the secure channel C2 is updated in both M and P.

Listing 6: Message type : key-update

```
<mes>
   <action type="key-update"/>
</mes>
```

3.4 Encrypted message format

The messages described in Section 3.3 are encrypted along with authenticated data `Auth Tag` using AES-GCM with a 256 bit key. The packet is depicted in Figure 5.

- IV: the 12 byte IV used for AES-GCM is added to the encrypted message. It is represented as a 24 byte hex-encoded string.
- AES-GCM encrypted data and Auth Tag: the messages from Section 3.3 are first zero-padded to a 256 byte block. The block is then encrypted and integrity protected using AES-GCM to generate a 256 byte cipher text and a 16 byte authentication tag. They are then hex-encoded into 512 bytes cipher text and 32 bytes authentication tag respectively. The ciphertext and the authentication tag are added to the communication packet as shown in Figure 5.

3.5 Approach

Start communication. To start the communication, run `/home/isl/t1/start.sh`. This starts RP and triggers the communication sequence shown in Figure 4.

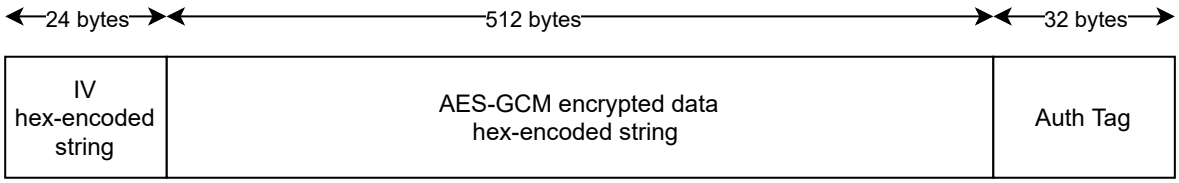

Figure 5: The messages from Section 3.3 are encrypted. This encrypted message is sent in the POST body of the HTTP packets exchanged between the different components.

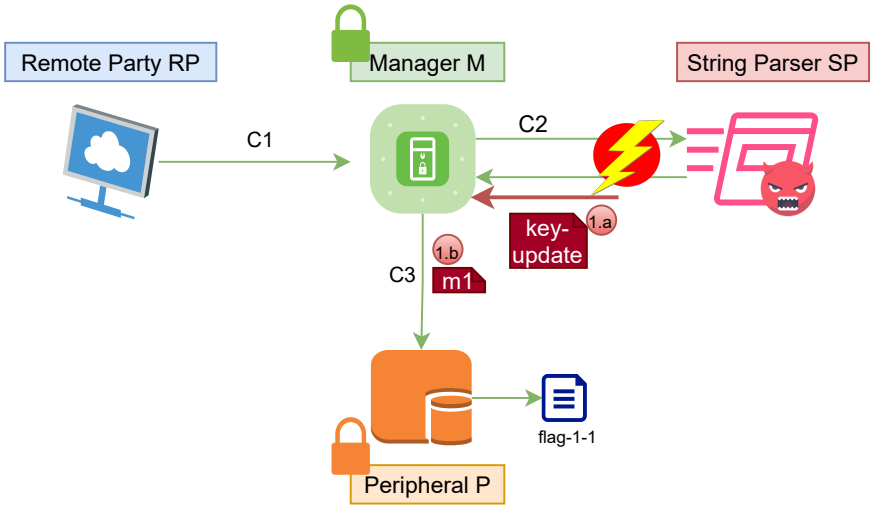

Figure 6: Attack the secure channel C2 and inject the `key-update` message into it. This tricks M into sending a message `m1` to P. On receiving `m1`, P outputs the flag

What to exploit? In this task we assume that the secure channels are implemented using safe cryptographic libraries and cannot be compromised using man-in-the-middle attacks over the network.

Instead you should think about the system capabilities of the attacker (see Section 1.1) and how it can be leveraged to compromise the execution of the processes. As stated in Section 1.1, you should use the capabilities of the attacker to compromise the channel C2 and the runtime execution of SP.

For Task 1.1, your attack on the channel C2 should trick M into sending a message `m1` to P as shown in Figure 6.

For Task 1.2, your attack on the runtime execution of SP should trick SP into sending `m2` to M. When M receives `m2`, it sends another message `m3` to P as shown in Figure 7.

Note that you do not know the contents of `m1`, `m2`, and `m3`; investigating this is not part of the task.

Extracting flags. To determine that you have successfully compromised channel C2 and the execution of SP, we have embedded two flags into P. We now explain how these flags can be extracted.

Task 1.1: If you successfully inject the `key-update` message into C2 and trick M into sending `m1` to P, P creates the flag file `flag-1-1` in `/home/isl/scripts` folder.

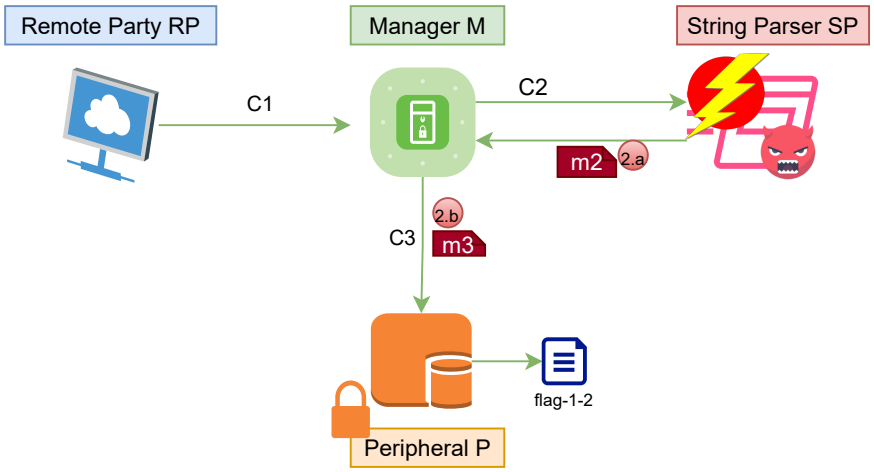

Figure 7: Compromise the runtime execution of SP and trigger the messages m2 and m3 to M and P respectively. On receiving m3, P outputs the flag.

Task 1.2: If you successfully compromise the runtime execution of SP to trigger the messages m2 and m3 as shown in Figure 7, P creates the flag file `flag-1-2` in `/home/isl/scripts` folder.

Tools. We have installed typical systems tools (e.g., Ghidra, gdb) to help you. For more details on how to use these tools please attend the exercise session and refer to the exercise handout on moodle.

4 Grading

4.1 Submission format

For this task you should implement the two end-to-end exploits in a single python file named `submit-1.py`. Please write your code using Python v3.8.10 that is pre-installed in the task VM. Please note that the only external Python library you can use in your script is `pycryptodome` i.e., your solution script cannot expect other libraries which are not part of Python's standard library. You should submit this python file on Moodle.

4.2 Grader program

Our grader program will run `submit-1.py`, read the `flag-1-*` files that the peripheral creates, and evaluate your solution based on the contents of the `flag-1-*` files. Specifically, our grader program downloads a fresh copy of the VM image that was provided to you and boots it. Next, it copies your script `submit-1.py` to `/home/isl/scripts` in the fresh VM it booted. It then executes `python3 submit-1.py >& /dev/null` on the VM and copies out the flag files (flag-1-1 and flag-1-2) from the `/home/isl/scripts` folder to our machine. It then reads the files and checks their content to calculate the final points for the task.

```
ubuntu@192:~/grader-1$ ./grade.sh submit-1.py &> out.txt
ubuntu@192:~/grader-1$ head -5 out.txt
"VM_Task1_Grade" {6260dcd4-96c0-403e-8e32-cd19e0774e72}
VM by name VM_Task1_Grade already exists. Not importing new VM. To import a fre
sh VM delete the existing VM first
Waiting for VM "VM_Task1_Grade" to power on...
VM "VM_Task1_Grade" has been successfully started.
copied solution script to VM
ubuntu@192:~/grader-1$ python3 check_flags.py 11-111-111
11-111-111
FLAG 1- EXPECTED: fd80176c50cfd9a5ef7586ac848e413a278f8ef7
FLAG 2- EXPECTED: ee2acaa6c0663a77b2a0b1f724f57b84e93076b3
Grading Task 1; Mat-Nr: 11-111-111
FLAG 1 READ: fd80176c50cfd9a5ef7586ac848e413a278f8ef7
T1_1: Correct Result - 10P
FLAG 2 READ: ee2acaa6c0663a77b2a0b1f724f57b84e93076b3
T1_2: Correct Result - 15P
Wrote grade file to ./grade-t1-11-111-111
ubuntu@192:~/grader-1$ cat grade-t1-11-111-111
Grading Task 1; Mat-Nr: 11-111-111
T1_1: Correct Result - 10P
T1_2: Correct Result - 15P
ubuntu@192:~/grader-1$
```

Figure 8: Example execution of the grader program

You can run this grader program on your system by downloading the grader-1.zip from Moodle. The grader program is distributed as a .zip file with the structure shown in Listing 7. Note that this grader program only tests for flags generated for ETH id `11-111-111`. For the final grading we use a similar grader program that tests different flag values for different students based on the ETH id in the `/home/isl/user` file.

Listing 7: Structure of grader-1.zip

```
1 grader-1.zip
2 ├──grade.sh
3 ├──check_flags.sh
4 └──README
```

Running the grader program yourself. Do not perform the following steps to run the grader program inside the task VM. You will have to use a Linux host machine with Ubuntu 20.04.3 LTS to run the grader program with VirtualBox installed. To run the grader, extract the `grader-1.zip` to `grader-1` folder on your host Linux machine. Next, follow the steps below:

- Download a fresh copy of the task VM from Moodle and copy it to `<path to grader-1>`. This is so that the grader program does not need to download the VM every time.
- Execute `chmod +x <path to grader-1>/grade.sh` to provide execute permissions for the scripts.
- Execute `<path to grader-1>/grade.sh <path to submit-1.py>`.
- Execute `<path to grader-1>/python3 check_flags.py <eth-id>` that checks

the flag files and grades them. The final grade file `grade-t1-<eth-id>` is created in the same directory with points for each flag. See Figure 8 for the final grade file format.

- Note that if you run the grader program multiple times, it will end up using the same VM. If you want to test it on a fresh VM, delete the grading VM named `VM_Task1_Grade` that was created by the grader program.

4.3 Writing your exploits

1. For this task, given the attacker capabilities in Section 1.1, the binaries for the components M, P and RP are treated as black boxes that cannot be reverse engineered. Trying to do this will not give you any useful results and is not a part of the task.
2. The flag files created by the peripheral P depend on your ETH student ID (e.g., `11-111-111`). You can put your student ID in the file `/home/isl/user`. However, please note that the example grader only works for the example student id `11-111-111`. So, if you use your own ID, the grader will fail—this is an expected outcome.
3. The peripheral we use for final grading uses a different approach to generate the flags based on individual student IDs. So hard coding the flags you get in your script or trying to reverse-engineer the flag generation will fail when we do the final grading.
4. The script you submit, `submit-1.py`, should not modify the flag files created by P.
5. The script you submit, `submit-1.py`, is responsible for starting the M, P, RP, SP processes. Ensure that you start M, P and SP before starting RP to guarantee correct operation. Our grader program will first check that these processes are running and then check the flag files.
6. The grader program terminates all scripts after 60 seconds.

4.4 Grading criteria

- **Task 1.1 [Points: 10]** For a successful flag-1-1, our auto-grader will give you 10 points.
- **Task 1.2 [Points: 15]** And for a successful flag-1-2, our auto-grader awards you 15 points.

We will only execute your submission file for a maximum of 60 seconds. After 60 seconds, our grader program will kill your script and then read the `flag-1-1` and `flag-1-2` files created by P to calculate the points for this task. If the `flag` files were not created by P, the grader program simply ignores the missing files. The grader computes the points for the `flag-1-*` files it found in the `/home/isl/scripts` folder.

Please note that we will use a clean environment for each script. Our grading VM will have the exact same environment as the VM that we gave you. The python script you submit should be self contained. We will not setup special execution environments (e.g., path variables) or debug your scripts. We will spawn a fresh VM to grade each submission. Do not assume that our grading VM will have:

1. any libraries or environment variables that are not available on the task VM.
2. sudo or root permissions for running your script.
3. internet to connect to other machines.
4. open ports to launch and execute other scripts, daemons, or services.
5. persistent storage on our VM or host machine.

The flag files (`flag-1-1` and `flag-1-2`) are auto-generated. Do not modify them. Any intentional attempts to write to these files will be detected and the grader program will terminate without evaluating your solution. Furthermore, any attempts to corrupt the machine our grader VM or host will result in zero points for this module. In addition, please carefully read Section 4.3 above.

5 Discussions

Friess Carl (carl.friess@inf.ethz.ch), Suter-Dörig Andris (sandris@student.ethz.ch), Mink Eric (minker@student.ethz.ch), Mark Kuhne (mark.kuhne@student.ethz.ch), Supraja Sridhara (supraja.sridhara@inf.ethz.ch) and Shweta Shinde (shweta.shivajishinde@inf.ethz.ch) will monitor the Moodle discussions. We encourage you to ask clarification questions on this public forum. That way everyone will benefit from clarifications we provide and it reduces duplicate questions. If you have a question that requires explaining your solution, please send a private email to all of us. Note that we will only answer questions that genuinely warrant a response. As a rule of thumb, we will not provide hints or help you to debug your code.

In conclusion, happy cracking & coding!